AADL Formal Analysis in Maude, PALS, and Synchronous AADL

José Meseguer

University of Illinois at Urbana-Champaign
1. Part I (Maude and AADL2Maude)
2. Part II (PALS and its Correctness)
3. Part III (Towards a Synchronous AADL Annex)
Maude and AADL to Maude

Acks: Joint work with Artur Boronat and Peter Ölveczky
Formal Model-Based Systems Engineering

Model-based systems development

- intuitive \textit{(graphical)} modeling language
- support for \textit{code generation} to go \textit{from model to code}
Model-based systems development

- intuitive (graphical) modeling language
- support for code generation to go from model to code
- need for precise semantics and formal analysis
Model-based systems development
- intuitive (graphical) modeling language
- support for code generation to go from model to code
- need for precise semantics and formal analysis

Formal model-based systems engineering
- define formal semantics for modeling language
Model-based systems development

- intuitive (graphical) modeling language
- support for code generation to go from model to code
- need for precise semantics and formal analysis

Formal model-based systems engineering

- define formal semantics for modeling language
- use code gen infrastructure to synthesize verification model
Model-based systems development

- intuitive (graphical) modeling language
- support for code generation to go from model to code
- need for precise semantics and formal analysis

Formal model-based systems engineering

- define formal semantics for modeling language
- use code gen infrastructure to synthesize verification model
- should allow system analysis without knowing target formalism
Real-Time Maude: formal analysis tool for real-time systems

- formalism emphasizes expressiveness and ease of specification
Real-Time Maude: formal analysis tool for real-time systems

- formalism emphasizes expressiveness and ease of specification
- object-oriented specification
  - nested objects
Real-Time Maude: formal analysis tool for real-time systems

- formalism emphasizes expressiveness and ease of specification
- object-oriented specification
  - nested objects
- high-performance simulation and model checking tool
Real-Time Maude: formal analysis tool for real-time systems

- formalism emphasizes expressiveness and ease of specification
- object-oriented specification
  - nested objects
- high-performance simulation and model checking tool
- successfully applied to advanced state-of-the-art systems
  - scheduling algorithms
  - wireless sensor network algorithms
  - embedded car software used by major car makers
  - 50+ page multicast protocol for active networks
  - ...
Real-Time Maude (II)

- Specification language:
  - algebraic equational specification for data types/functions
  - rewrite rules define instantaneous transitions
  - tick rewrite rules define time elapse
Real-Time Maude (II)

- Specification language:
  - algebraic **equational specification** for data types/functions
  - **rewrite rules** define instantaneous transitions
  - **tick rewrite rules** define time elapse

- Analysis capabilities:
  - **rewriting** for simulation/prototyping
  - **explicit-state** breadth-first **search** for **reachability analysis**
  - **LTL model checking**
industry standard for embedded/RT systems modeling


avionics, aerospace, medical devices, robotic, . . .
**AADL**

- **industry standard** for embedded/RT systems modeling
- avionics, aerospace, medical devices, robotic, . . .
- **hierarchy of software and hardware** components
  - process, thread, subprogram, . . .
  - processor, memory, device, bus, . . .
- **OSATE**: Eclipse plug-ins for AADL
AADL

- **industry standard** for embedded/RT systems modeling
- avionics, aerospace, medical devices, robotic, . . .
- hierarchy of software and hardware components
  - process, thread, subprogram, . . .
  - processor, memory, device, bus, . . .
- **OSATE**: Eclipse plug-ins for AADL
- no formal semantics!
Targeted Behavioral Subset

- Focus on **software** components
  - hierarchical components
  - connections, data/event ports,
  - thread, process, systems, ...
  - mode transitions
  - events
  - periodic, aperiodic, sporadic dispatches
- Thread behaviors: **Behavior Annex**
  - transition systems with local state variables
- **Hardware** components and **scheduling** not targeted yet!
Object-oriented semantics
“Line-by-line” translation
“One-to-one” correspondence AADL model $\leftrightarrow$ Real-Time Maude term
- hierarchical objects
- easy to map counterexamples to AADL behaviors
Class for any software component:

```maude
class Component | features : Configuration,
    subcomponents : Configuration,
    properties : Properties,
    connections : ConnectionSet,
    modes : Modes,
    inModes : ModeNameSet .
```

Class for any software component:

```plaintext
class Component | features : Configuration,
                 subcomponents : Configuration,
                 properties : Properties,
                 connections : ConnectionSet,
                 modes : Modes,
                 inModes : ModeNameSet .
```

Thread components class:

```plaintext
class Thread | behavior : ThreadBehavior,
             status : ThreadStatus,
             deactivated : Bool .
subclass Thread < Component .
```
AADL declaration

thread taThread
  features pressEvent: out event data port Behavior::integer;
  properties Dispatch_Protocol => periodic;   Period => 1 sec;
end taThread;

thread implementation taThread.impl
  annex behaviorSpecification {**
    states        s0: initial complete state;
    transitions   s0 -> s0 {pressEvent!(1);};
  **};
end taThread.impl;

corresponds to Real-Time Maude declarations ...
eq thread(taThread) = features (pressEvent out event data thread port) 
        properties DispatchProtocol(Periodic); Period(1 Sec).

eq INSTANCE-NAME thread taThread . impl in modes MNS 
   = < INSTANCE-NAME : Thread | 
        modes : noModes, 
        inModes : MNS, 
        features : features(thread(taThread)), 
        subcomponents : none, 
        connections : none, 
        properties : properties(thread(taThread)), 
        behavior : states initial: s0 complete: s0 
            transitions s0 -[]-> s0 {(pressEvent ! (1))}
   > .
Rewrite rules defining dynamics:

\[ rl \ [\text{periodic-dispatch}] : \]
\[
< 0 : \text{Thread} \mid \text{properties} : \text{periodic-dispatch}(T, 0) \ \text{PROPS}, \\
\quad \text{status} : \text{completed}, \\
\quad \text{features} : \text{PORTS} > \\
\]
\[
=> \\
< 0 : \text{Thread} \mid \text{properties} : \text{periodic-dispatch}(T, T) \ \text{PROPS}, \\
\quad \text{status} : \text{active}, \\
\quad \text{features} : \text{dispatchInputPorts}(\text{PORTS}) > .
\]

and ...
Have enriched OSATE with generation of Real-Time Maude verification model from AADL design model
Case Studies

- Safe interoperation of ventilator, X-ray machine, and controller
  - system by Lui Sha (UIUC)
  - AADL model by Min Young Nam (UIUC)
- Active standby *avionics* example
  - system by Steve Miller (Rockwell-Collins)
  - AADL model by Abdullah al-Nayeem (UIUC)
Concluding Remarks

- **Real-Time Maude semantics, simulation, and formal analysis** for a behavioral subset of AADL
  - Real-Time Maude expressiveness key
  - equations and rules
  - AADL model $\simeq$ Real-Time Maude model
Concluding Remarks

- **Real-Time Maude semantics, simulation, and formal analysis** for a behavioral subset of AADL
  - Real-Time Maude expressiveness key
  - equations and rules
  - AADL model $\simeq$ Real-Time Maude model
- Generation of **Real-Time Maude** model using OSATE
- Combine convenience of **AADL modeling** and **Real-Time Maude analysis**
Concluding Remarks

- **Real-Time Maude semantics, simulation, and formal analysis** for a behavioral subset of AADL
  - Real-Time Maude expressiveness key
  - equations and rules
  - AADL model $\simeq$ Real-Time Maude model

- Generation of **Real-Time Maude** model using OSATE
- Combine convenience of **AADL modeling** and **Real-Time Maude analysis**

- Future work:
  - larger subsets of AADL
  - “formal property annex” in AADL
  - synchronous AADL annex
Acks:

- Formal model and correctness is joint work with Peter Ölveczky
- This is part of broader PALS collaboration with:
  - Peter Ölveczky at University of Oslo
  - Steve Miller and Darren Cofer at Rockwell-Collins
  - Lui Sha, Abdullah Al-Nayeem, and Mu Sun at UIUC
Motivation

- Many **distributed real-time systems**: collection of components that communicate asynchronously, each component has a local clock, should change state and respond to environment input within hard real-time bounds, should achieve **virtual synchrony**

- **Examples**: integrated modular avionics, distributed control in motor vehicles, interoperation of medical devices

- Often **safety-critical** systems

- Design, verification, and implementation hard and error-prone: asynchronous communication, message delays, clock skews

- Model checking verification often unfeasible due to the state space explosion caused by the system’s concurrency
Formal Architectural Patterns

Formal architectural patterns:

- Generic formal specification of an engineering solution to a generic design problem
  - formal correctness guarantees
  - reduces system complexity
    - verification and correctness of implementation much easier

- Amortize verification effort on a general family of designs
**PALS**: Physically Asynchronous, Logically Synchronous

- Reduces design and verification of a distributed real-time system (that should ensure virtual synchrony and satisfy hard real-time bounds) to its underlying synchronous version
- Relies on **asynchronous bounded delay (ABD)** network infrastructure
  - max bound on the communication delay
- Assumes underlying **clock synchronization** guarantees maximal bound on the clock skews
PALS can be seen as a formally verified model transformation

\[(E, \Gamma) \mapsto A(E, \Gamma)\]

- \(E\) is a synchronous design
- \(\Gamma\) are performance bounds on
  - max clock skew \(\epsilon\) between any local clock and a “global” clock
  - min and max time for reading input, performing a “transition,” and producing output
  - min and max message transmission delay
- \(A(E, \Gamma)\) is the corresponding asynchronous design of the distributed real-time system
We have verified the PALS transformation

$$(\mathcal{E}, \Gamma) \mapsto A(\mathcal{E}, \Gamma)$$

- Synchronous design $\mathcal{E}$ formalized as an ensemble of typed machines
- $A(\mathcal{E}, \Gamma)$ formalized in Real-Time Maude
- Verified the bisimulation between $(\mathcal{E}, \Gamma)$ and $A(\mathcal{E}, \Gamma)$
- Proved optimality of the period of $A(\mathcal{E}, \Gamma)$
A synchronous model is an ensemble of state machines.

In the synchronous composition of an ensemble, in each “iteration,” each machine at the same time:

- reads input from environment and from the values generated in the previous round in the “feedback” loops
- produces output and changes its local state according to its local transition relation

The environment can be abstractly seen as nondeterministically generating arbitrary outputs satisfying a given constraint.
In the distributed implementation $A(\mathcal{E}, \Gamma)$, adds a “wrapper” around each state machine in $\mathcal{E}$

- each wrapper has an input buffer that stores messages arrived during the round
- each wrapper has an output buffer to hold outgoing messages until they can be sent to the network
The behavior of each node is:

- **at the beginning** of each **PALS period** (given by **local clock**):
  - read messages from input buffer
  - execute local transition (change local state and generate new messages)
  - new messages are put in output buffer with a **back-off delay**

- messages from the output buffer are sent to the network when the backoff timer has expires **and** the execution of the local transition has finished

- received messages are put in the input buffer
Some Assumptions

- **External clock synchronization**
  - difference between “local clock” time and “real” (global) clock time is always less than $\epsilon$
- **Local clock**: $c_j : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$
  - monotonic and continuous
  - $|c_j(x) - x| < \epsilon$ for all “real” times $x$
- Time for (processing input + executing transition + generating output) $\in [\alpha_{\text{min}}, \alpha_{\text{max}}]$ with $0 \leq \alpha_{\text{min}} \leq \alpha_{\text{max}}$
- Message transmission time $\in [\mu_{\text{min}}, \mu_{\text{max}}]$ with $0 \leq \mu_{\text{min}} \leq \mu_{\text{max}}$
The smallest possible period $T$ is

$$\mu_{max} + 2 \cdot \epsilon + \max(2 \cdot \epsilon - \mu_{min}, \alpha_{max})$$

We have proved optimality of $T$
A state in $A(\mathcal{E}, \Gamma)$ is stable iff all input buffers are full, all output buffers are empty, and there are no messages in transit.

The function $\text{sync}$ maps stable states in $A(\mathcal{E}, \Gamma)$ to states in $\mathcal{E}$ in the obvious way.

Can define Kripke structures $(\mathcal{E}_{ce}, L)$ for $\mathcal{E}$ (with environment constraint $c_e$) and $(A(\mathcal{E}, \Gamma), L')$ in the expected way.
Main Correctness Result

Theorem

Given a formula \( \varphi \in CTL^*(AP) \), and a state predicate \( \text{stable} \notin AP \) characterizing stable states, there is a formula \( \varphi_{\text{stable}} \in CTL^*(\{\circ\}(AP \cup \{\text{stable}\})) \) defined as follows:

\[
\begin{align*}
\text{a}_{\text{stable}} & = a, \text{ for } a \in AP \\
\neg \varphi_{\text{stable}} & = \neg(\varphi_{\text{stable}}) \\
(\varphi_1 \land \varphi_2)_{\text{stable}} & = \varphi_{1_{\text{stable}}} \land \varphi_{2_{\text{stable}}} \\
(\varphi_1 \lor \varphi_2)_{\text{stable}} & = (\text{stable} \rightarrow \varphi_{1_{\text{stable}}} \land \varphi_{2_{\text{stable}}} \land \varphi_{\text{stable}}) \\
(\circ \varphi)_{\text{stable}} & = \text{stable} \land (\neg \text{stable} \land (\neg \text{stable} \land \varphi_{\text{stable}})) \\
(\forall \varphi)_{\text{stable}} & = \forall \varphi_{\text{stable}}
\end{align*}
\]

such that for each reachable stable state \( s \) in \( A(\mathcal{E}, \Gamma) \) we have

\[
(A(\mathcal{E}, \Gamma), L'), s \models \varphi_{\text{stable}} \iff (\mathcal{E}_{ce}, L), \text{sync}(s) \models \varphi,
\]

where \( L' : T_{A(\mathcal{E}, \Gamma)_{\text{globalSystem}}} \rightarrow \mathcal{P}(AP \cup \{\text{stable}\}) \) is a labeling function satisfying \( L'(s) = L(\text{sync}(s)) \cup \{\text{stable}\} \) when \( s \) is a stable state, and \( \text{stable} \notin L'(s) \) otherwise.
In integrated modular avionics, there are for fault tolerance properties multiple cabinets (with power supply, computer, I/O, etc.) distributed in an aircraft.

**Active Standby System**: decide which cabinet is active
- standby side should monitor the side's functionalities and the pilot’s manual switch
- standby side notifies active side if change of active sides needed

**Synchronous design** must be realized for the distributed cabinets in the aircraft.

Our models based on AADL model by Abdullah Al-Nayeem of a similar spec developed by Steve Miller and Darren Cofer.
Architecture of active standby for two sides:
We defined the **synchronous model** of active standby in Maude.

We defined a **very simplified PALS asynchronous model** in (Real-Time) Maude:
- execution times 0
- perfect clocks
- two models of message delays:
  - version 1: no message delay
  - version 2: message delay 0 or 1
- discrete time domain
- shortest possible PALS period

Execution times for **model checking** an invariant:

<table>
<thead>
<tr>
<th>Model</th>
<th>Max.msg.dly</th>
<th># states</th>
<th>ex.time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchr.</td>
<td>n/a</td>
<td>185</td>
<td>0.2 sec.</td>
</tr>
<tr>
<td>Asynchr.</td>
<td>0</td>
<td>3,047,832</td>
<td>2000 sec.</td>
</tr>
<tr>
<td>Asynchr.</td>
<td>1</td>
<td></td>
<td>aborted</td>
</tr>
</tbody>
</table>
Requirement of Active Standby

The active standby system should satisfy the following requirements:

\( R_1 \): Both sides should agree on which side is active (provided neither side has failed, the availability of a side has not changed, and the pilot has not made a manual selection).

\( R_2 \): A side that is not fully available should not be the active side if the other side is fully available (again, provided neither side has failed, the availability of a side has not changed, and the pilot has not made a manual selection).

\( R_3 \): The pilot can always change the active side (except if a side is failed or the availability of a side has changed).

\( R_4 \): If a side is failed the other side should become active.

\( R_5 \): The active side should not change unless the availability of a side changes, the failed status of a side changes, or manual selection is selected by the pilot.
Have verified improved versions of the five desired properties of (the synchronous version of) active standby:

- each model checking took about 0.8 seconds

Example (Requirement 4):

If a side is failed the other side should become active

\[
\text{eq } R4 = \square \left( (\text{side } 1 \text{ failed } \land \neg \text{ side } 2 \text{ failed}) \rightarrow 0 (\neg \text{ side } 2 \text{ failed } \rightarrow \text{ side } 2 \text{ active}) \right) \\
\land (\text{side } 2 \text{ failed } \land \neg \text{ side } 1 \text{ failed}) \rightarrow 0 (\neg \text{ side } 1 \text{ failed } \rightarrow \text{ side } 1 \text{ active}) \right)
\]

Maude> (red modelCheck(init, R4) .)
rewrites: 101597 in 825ms cpu (831ms real)
result Bool : true
Towards a Synchronous AADL Annex

Acks: Joint work with Abdullah Al-Nayeem, Kyungmin Bae, Mingyoug Nam, Peter Ölveczky, and Lui Sha.
Thanks: to SEI AADL Group for very helpful discussions.
Why Synchronous AADL?

- Exploit the PALS pattern
- Define high-level synchronous designs in AADL
- Verify synchronous designs with “synchronous-AADL-to-Maude” tool
- Use PALS to automatically generate asynchronous distributed AADL model from synchronous AADL design model
What Should “Synchronous AADL” Look Like?

- Subset of AADL extended with **new annotations**
- Focus on AADL subset to model synchronous PALS designs
  - abstract from hardware and memory, etc.,
  - keep some real-time features
  - focus on behavioral and structuring subset
- How to **trigger** a synchronous step
  - time-triggered
  - event-triggered
All threads have the same period

- execute synchronously at the start of the period
- same-level connections are delayed connections
Instead of only delayed connections, the system might want to react to input from the environment immediately

- Immediate connection from (single) dispatcher
- Other connections are delayed connections
- Non-dispatcher components contain aperiodic threads
Behaviors

- Thread transitions in AADL behavior annex can be nondeterministic, and also in PALS synchronous models.
- The behavior of components, except for the environment, is most often deterministic.
- Assumption of determinism of certain components could make model checking much more efficient.
- Language support for declaring threads to be “nondeterministic” (or “environment”).
AADL Language Extensions

Use properties construct in AADL to declare synchronous systems in the component (implementation) that contains the synchronous components.

Recall active standby architecture:
Example: Time-Triggered Dispatch

system ActiveStandbySystem
end ActiveStandbySystem;

system implementation ActiveStandbySystem.impl

subcomponents
    sideOne: system Side1::Side1.impl;
    sideTwo: system Side2::Side2.impl;
    env: system Environment::Environment.impl;

connections
    C1: data port sideOne.side1ActiveSide \rightarrow>> sideTwo.side1ActiveSide;
    C2: data port sideTwo.side2ActiveSide \rightarrow>> sideOne.side2ActiveSide;
    C3: data port env.manualSelection \rightarrow>> sideOne.manualSelection;
    C4: data port env.manualSelection \rightarrow>> sideTwo.manualSelection;
    C5: data port env.side1Failed \rightarrow>> sideOne.side1Failed;
    C6: data port env.side2Failed \rightarrow>> sideTwo.side2Failed;

properties
    SynchAADL::Synchronous => periodic(1ms)
end ActiveStandbySystem.impl;
system ActiveStandbySystem
end ActiveStandbySystem;

system implementation ActiveStandbySystem.impl
  subcomponents
    sideOne: system Side1::Side1.impl;
    sideTwo: system Side2::Side2.impl;
    env: system Environment::Environment.impl;
  connections
    C1: data port sideOne.side1ActiveSide ->> sideTwo.side1ActiveSide;
    C2: data port sideTwo.side2ActiveSide ->> sideOne.side2ActiveSide;
    C3: data port env.manualSelection -> sideOne.manualSelection;
    C4: data port env.manualSelection -> sideTwo.manualSelection;
    C5: data port env.side1Failed -> sideOne.side1Failed;
    C6: data port env.side2Failed -> sideTwo.side2Failed;
  properties
    SynchAADL::Synchronous => aperiodic with dispatcher env;
end ActiveStandbySystem.impl;
Research Tasks

- Define **AADL synchronous annex**
- Define Maude semantics for synchronous AADL models
- Develop optimized model checking techniques for such models
- Use OSATE code generation infrastructure and execution environment for AADL to synthesize a Maude verification model from a synchronous AADL design model, and to integrate Maude verification into OSATE
- Define automatic model transformations from synchronous AADL models to distributed asynchronous AADL models